International Journal of Engineering Inventions
e-ISSN: 2278-7461, p-ISSN: 2319-6491
Volume 3, Issue 12 (July 2014) PP: 01-07

A Study of Significant Software Metrics

Neha Saini[1], Sapna Kharwar[2], Anushree Agrawal[3]
[1]*Department of software Engineering, Delhi technological University, Delhi*
[2]*Department of software Engineering, Delhi technological University, Delhi*
[3]*Department of software Engineering, Delhi technological University, Delhi*

Abstract: *A software system continues to grow in size and complexity, it becomes increasing difficult to understand and manage. Software metrics are units of software measurement. As improvement in coding tools allow software developer to produce larger amount of software to meet ever expanding requirements. A method to measure software product, process and project must be used. In this article, we first introduce the software metrics including the definition of metrics and the history of this field. We aim at a comprehensive survey of the metrics available for measuring attributes related to software entities. Some classical metrics such as Lines of codes LOC, Halstead complexity metric (HCM), Function Point analysis and others are discussed and analyzed. Then we bring up the complexity metrics methods, such as McCabe complexity metrics and object oriented metrics(C&K method), with real world examples. The comparison and relationship of these metrics* are *also presented.*

Keywords: *Software metrics, software measurement scales, Function points, software attributes, categories of metrics, metrics for measuring internal attributes, software complexity.*

I. Introduction

In the field of science everything begins with quantification. All engineering disciplines have metric and some measurable quantity, so in the field of computer science also one needs some quantity to measure. Software measurement helps us in answering few questions like:-

- How good is the design
- How complex is the code
- How much efforts will be required

The above discussion makes it clears that software measure help in decision making in various life cycle phases. Software metrics can be defined as the continuous application of measurement based techniques to the software development process and its products, to supply meaning full and timely management information. Some definitions proposed for software metrics are [20]

Definition 1: Software Metrics provide a measurement for the software and the process of software production. It is giving quantitative values to the attributes involving in the product or the process.
Definition 2: Software metrics is to give the attributes some quantitative descriptions. These attributes are extracting from the software product, software development process and the related resources. They are product, process and resources.
Definition 3: Software measurement provides continuous measures for the software development process and its related products. It defines, collects and analyzes the data of measurable process, through which it facilitates the understanding, evaluating, controlling and improving the software product procedure.
Definition 4: According to IEEE "standard of software Quality Metrics Methodology", software metrics is a function, with input as the software data, and output is a value which could decide on how the given attribute affect the software.
Definition 5: According to J A McCall "The metrics are quantitative measure of the characteristics of the software which provide certain qualities. The hierarchical structure of the frame work provides relevance to management at one level and to the software developer at other level." [1, 20]

In 1979, Albrecht gave function points metrics, based on the requirement specification. Thereafter, in 1993 J-Y Chen and J-F Liu proposed a method, which used parameters like complexity, attributes, and reusability of class. to measure the Object- oriented software[18]. In 1994, Chidamber Kemerer based on inheritance tree gave a set of object-oriented metrics known as CK suite [6, 16,18]. In 1995, Brito also proposed, a set of metrics based on object-oriented attributes, called MOOD metrics [17]. In 2001, Victor and Daily based on software components gave a method called as SPECTRE, which is used to estimate time and modules level [20]. In 2003, Hastings and Sajeev proposed Vector Size Measure (VSM) metrics for early stage measurements in the software lifecycle. Subsequently, Arlene F. proposed a forecast metric based on the objects and attributes, it was used estimate work load and production power [19, 20].

1.1.Categories of metrics

Metrics can be classified into three categories [1]

- Product Metrics
- Process Metrics
- Project Metrics

1.1.1 Product Metric: Software Product metrics are measures of software products such as source code and design documents. Software product metrics that measure software product in different paradigms are also different. In procedural paradigm metr ics measure functions and how functions interact, In Object oriented paradigm classes and how classes interact is measured. [5]. Product Metrics can be further classified as shown in Fig 1[1, 5].

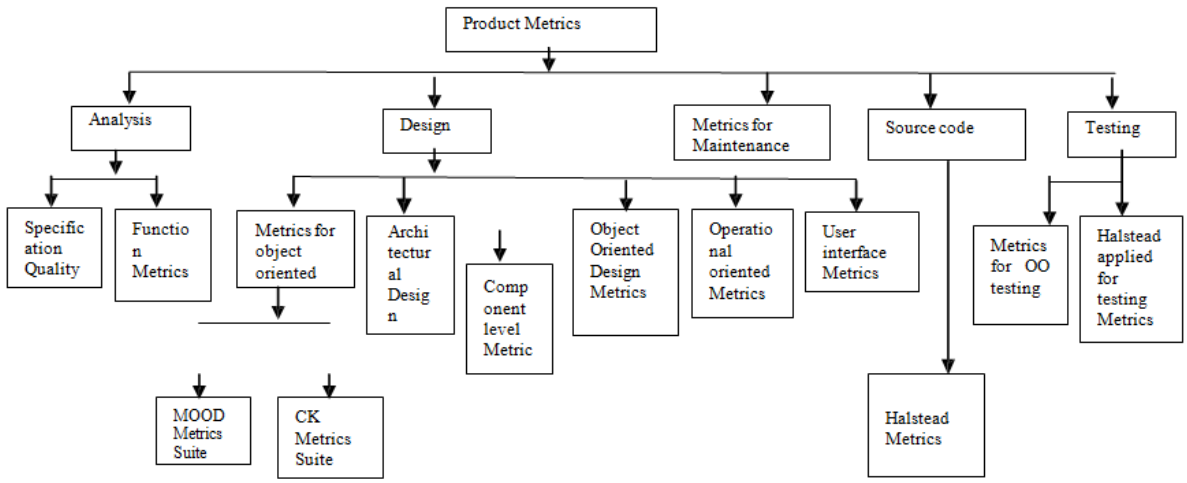

Fig 1. Classification of Product Metrics [2]

1.1.2 Process: Process metrics emphasize on the process of the software development. It mainly focus on how long a process last, what about the cost, whether the methods used are effective etc.. It includes the improvement of the process and the prediction for the future process. The main part of this metrics includes maturity, management, life cycle, product ratio, defect ratio, etc.. It is beneficial to the control and management of the whole development procedure.

1.1.3 Project: Project metrics are designed to control the project situation and status. The metrics includes scale, cost, workload, status, production power, risk, the degree of satisfaction from clients, etc.. Project metrics is used to analyze the project to avoid the risk factors and help to optimize the development plans. Afterward, project metrics improves the quality of the product via advancement in techniques, methods and management strategies.

II. Software Measurement

In order to measure, one needs to identify an entity and a specific attribute of it. It is very important to define in a clear way what one is measuring because otherwise one may not be able to perform the measure or the measures obtained can have differen t meaning to different people. Measurement can take place in all the different phases of software development: Requirement Analysis, Specification, Design, Coding and Verification. Undoubtedly, measurement is most useful if carried out in the early phases. Different internal and external attributes are shown in Table 1.

Table I. Entities and attributes [3]

Entity	**Internal Attributes**	**External Attributes**
A. **Product**		
Requirements	Size, Reuse, Modularity, Redundancy, Functionality	Understandability, Stability
Specification	Size, Reuse, Modularity, Redundancy, Functionality	Understandability, Maintainability
Design	Size, Reuse, Modularity, Coupling, Cohesion	Comprehensibility, Maintainability, Quality

Code	Size, Reuse, Modularity, Coupling, Cohesion, Control Flow Complexity	Reliability, Usability, Reusability, Maintainability
Test set	Size, Coverage level	Quality
B. **Process**		
Requirements Analysis	Time, Effort	Cost effectiveness
Specification	Time, Effort, Number of requirements Changes	Cost effectiveness
Design	Time, Effort, Number of specification Changes	Cost effectiveness
Coding	Time, Effort, Number of design Changes	Cost effectiveness
Testing	Time, Effort, Number of code changes	Cost effectiveness
C. **Resource**		
Personnel	Age, Cost	Productivity, Experience
Team	Size, Communication Level, Structure	Productivity
Software	Size, Price	Usability, Reliability
Hardware	Price, Speed, Memory size	Usability, Reliability

Many metrics have been proposed for structural complexity and they measure a number of internal attributes of software. Structural metrics can be divided in intra module metrics and inter module metrics. Module metrics are focused at the individual module level (subprogram or class) and comprehend: size metrics, control flow complexity metrics, data structure metrics and cohesion metrics. Inter module metrics measure the interconnections between modules of the system and are constituted by coupling metrics. Various measuring scales with example are written in table 2.

Table- II. Measurement scales [2]

Measurement scales	**Definition**	**Example**
Nominal	Items are assigned to group or categories, it is diffrentiative, no quantitative information is generated, no ordering is implied.	Recursive or non-recursive program Types of errors Binary executable program, or DLL
Ordinal	Measurements are ordered, higher no represent higher vales but the no are only for ordering	CMM maturing levels How often software fails
Interval	It separate classes based on vale you know exactly when the item crosses one class and goes to another. Classes are ordered. Additional and subtraction can be performed but	Logs of event on dates.
Ratio	Has ordering, interval sizes and ration are possible. Value	LOC (Length of code) as statement count.
Absolute	Counting entities in the entity set. All arithmetic operation are meaning full. There is only one	Lines of code, No of failures, no of project engineers.

III. Standard Metrics

Metrics chosen among the most widely used and are grouped according to the phase of software development in which they can be applied. The overview of various metrics is presented below.

a. Lines of code

The simplest software metric is the number of lines of code. But it is not clear whether one have to count the comments as well, even if they give a big contribution to the understand ability of the program, and the declarative parts, that give a contribution to the quality of the program [2,5]. Also some languages allow more than one instruction on the same line. If one counts the number of instructions, then we are still uncertain about comments and declarations. "A line of code is any line of program text that is not a comment or blank line, regardless of the number of statements or fragments of statements on the line. This specifically includes all lines containing program headers, declarations, and executable and non-executable statements." One of the main disadvantages with LOC is that it does not take into account the goodness of the

code: if one uses LOC to measure productivity, a short well designed program is "punished" by such a metric. Another disadvantage is that it does not allow comparing programs written in different languages.

In spite of these problems, lines of code are a widely used metric due to its simplicity, ease of application, inertia of tradition and absence of alternative size measures. Moreover, there are many empirical studies that demonstrate the usefulness of LOC:

LOC have been used for a variety of tasks in software development: planning, monitoring the progress of projects, predicting. From the point of view of measurement theory, LOC are a valid metric for the length attribute of a program because the empiri cal relation "is shorter than" is perfectly represented by the relation between lines of code [5]:

x "is shorter than" y -- LOC(x) < LOC(y)

b. Halstead's Software Science

M. Halstead is based on the assumption that a program is made only of operators and operands and that the knowledge of the numbers of distinct and repeated operators and operands is sufficient for determining a number of attributes of software such as program length, volume, level, programming effort. It is interesting to note that, even if the equation provides only estimates, they are exact, not statistical. Here let us assume

Operand: every variable or constant present in the program

Operator: every symbol or combination of symbols that influences the value or order of an operand.

Punctuation marks, arithmetic symbols (such as +,-,* and /), keywords (such as if, while, do, etc.), special symbols (such as:=, braces, parenthesis, ==, !=) and function names are operators. Some attributes are considered fundamental and used to derive all the other attributes of the model:

n1 the number of distinct operators n2 the number of distinct operands N1 is total number of operators

N2 is total number of operands

The length N of a program is the total number of symbols in the program and is given by

N=N1+N2 (1) Expected software length:

H= n1 log2(n1) + n2 log2 (n2)

Volume: V= N*log2(n)

Level: L=V*/V= (2/n1)*(n2/N2) Difficulty: D=V/V*= (n1/2)*(N2/n2)

Effort: E= V*D (2)

c. High-Level Design metrics

High-level designs is either traditional or object oriented. The goal of these metrics is to assess the quality of a software design with respect to its error proneness. The high level design of a system is seen as a collection of modules. A module is a provider of a computational service and is a collection of features, i.e. constants, type, variable and subroutine definition s. It is an object in object oriented systems, but can be present as well in traditional systems. Modules are composed of two 18 parts: an interface and a body (which may be empty). The interface contains the computational resources that the module makes visibl e for use to other modules. The body contains the implementation details that are not to be exported. The high-level design of a system consists only in the definitions of the interfaces of modules. A software parts is a collections of modules. The definitions of interactions.

Data declaration-Data declaration (DD) Interaction A data declaration A DD-interacts with another data declaration B if a change in A's declaration or use may cause the need for a change in B's declaration or use. Data declaration-Subroutine (DS) Interaction A data declaration DS-interacts with a subroutine if it DD-interacts with at least one of its data declarations.

Let us consider now the attribute cohesion. It is the extent to which features that are conceptually related belong to the same module. It is desirable to have a high cohesion because otherwise we would have features that depend on each other scattered all over the system, with the result that the software could be more error prone. The set of cohesive interactions in a module m is the union of the sets of DS interactions and DD-interactions, with the exception of those DD-interactions between a data declaration and a subroutine formal parameter.

d. McCabe's Cyclomatic Complexity

It concentrates on control flow complexity and does not take into account, the contribute to complexity that derives from data [9]. A program control flow can be represented by a graph which has a unique entry node and exit node, and in which all nodes are reachable from the entry and the exit is reachable from all nodes. Idea is to measure the complexity by considering the number of paths in the control graph of the program. But even for simple programs, if they contain at least one cycle, the number of paths is infinite. Therefore he considers only the number of independent paths: these are complete paths, (paths that go from the starting node to the end node of the graph), such that their linear combinations can produce all the set of complete path of a program.

e. Function Points

Function Points give a measure of the functionality of the system starting from a description, in natural language, of user's requirements. Thus they provide a technology independent estimate of the size of the final program and are probably the only measure of size that is not related to code. The measurement of function points is based on identifying and counting the functions that the system has to perform. Function Points are computed in terms of

- No of external inputs
- No of external outputs
- No of external inquiries
- No of external files
- No of internal files

Each function identified in the system is then classified to three levels of complexity: simple, average and complex. According to the complexity and the function type, a weight is assigned to each function and all the weights are summed up to give the unadjusted function point count. The final function point count is then obtained by multiplying the unadjusted count by an adjustment that expresses the influence of 14 general system characteristics.

3.1 Object oriented metrics

3.1.1. MOOD metrics suite

f. Method Hiding Factor (MHF) and Attribute Hiding Factor (AHF)

These metrics are basically measures of encapsulation [6, 16, 17]. In the encapsulation, MHF and AHF act as measures of "the use of the information hiding concept". MHF is defined formally [17, 18] as:

$$\frac{\sum_{i=1}^{TC} \sum_{m=} M(Ci) (1-V(M}{\sum_{i=1}^{TC} (M_d(C_i)_{mi}))} \quad (3)$$

Where M_d (C_i) is no of methods in class, TC is total No of classes and,

$$V(M_{mi}) = \frac{\sum_{i=1}^{TC} (\text{is visible } (M_{mi}, C_j))}{TC-1} \quad (4)$$

For all classes, C_1, C_2, ..., C_n, a method counts is 0 if it can be used by another class, and 1 if it cannot. The total for the system is divided by the total number of methods, to give the percentage of hidden methods in the system. AHF was defined in similar way, but using attributes rather than methods. The definitions of MHF and AHF cause discontinuities for systems with only one class. Data encapsulation, Information hiding is very primitive in today's open world. Both MHF and AHF use code visibility to measure information hiding, and thus are validated using all validation criteria. MHF and AHF measure the relative *amount* of information hiding and not the *quality* of the information hiding design decisions.

g. Method Inheritance Factor (MIF) and Attribute Inheritance Factor (AIF) Metrics

MIF and AIF measure the number of inherited methods and attributes respectively as a proportion of the total number of methods/attributes [6, 16]. There is a relationship between the relative amount of inheritance in a system and the number of methods/attributes which have been inherited. It can be defined as:-

$$\frac{\sum_{i=1}^{TC} M_i(Ci)}{\sum_{i=1}^{TC} M_a(C_i)} \quad (5)$$

here, $M_a(C_i) = M_d(C_i) + M_i(C_i)$

$Md(C_i)$ = the number of methods declared in a class,

$Ma(C_i)$ = the number of methods that can be invoked in association with C_i,

$Mi(C_i)$ = the number of methods inherited in C_i.

The total for the system is divided by the total number of methods, including any which have been inherited.

h. Coupling Factor (CF)

This metric measures the coupling between classes, excluding coupling due to inheritance. CF has been defined [16, 17] as:

$$\frac{\sum_{i=1}^{TC} \sum_{j=1}^{TC} (\text{is client } (C_i, C_j))}{TC^2-TC} \quad (6)$$

Here, is client (C_i, C_j)=1 iff C_c => C_s ^ C_c =/C_s otherwise equal to 0

Cc =>Cs represents the relationship between a client class, *Cc*, and a supplier class, *Cs.*
CF is calculated by considering all possible pair wise sets of classes, and asking whether the classes in the pair are related, either by message passing or by semantic association links (reference by one class to an attribute or method of another class). These relationships are considered to be equivalent as far as coupling is concerned [7], [9]. Thus, CF is a direct measure of the size of a relationship between two classes, for all pair wise relationships between classes in a system. There are two possible approaches to validating CF. Firstly, CF can be considered as *direct* measure of interclass coupling. Alternatively, one can consider CF to be an *indirect* measure of the attributes to which it was said to be related [8]. CF will enable us to distinguish between different programs with different levels of coupling. A system with a high level of interclass coupling will have a high CF value.

i. Polymorphism Factor (PF)
It is a measure of polymorphism potential. It can be defined as [6, 16]:

$$\frac{\sum_{i=1}^{TC} M_o(C_i))}{\sum_{i=1}^{TC} [(M_n(C_i) * DC(C_i)]} \quad (7)$$

Here $M_d(C_i) = M_n(C_i) + M_o(Ci)$ and
Mn(*Ci*) = number of new methods,
Mo(*Ci*) = number of overriding methods,
DC(*Ci*) = descendants count

PF is the number of methods that redefine inherited methods, divided by the maximum number of possible distinct polymorphic situations PF is an indirect measure of relative amount of dynamic binding in a system.

3.1.2 CK metric suite*:*
j. Weighted Method per class (WMC)
It can be defined as consider a Class S_1, with methods $M_1 \ldots\ldots M_n$ defined in class. Let $C_1 \ldots. C_n$ be the complexity of the methods then[6, 16].

$$WMC = \sum_{i=1}^{n} C_i \quad (8)$$

WMC relates the complexity of the things as methods are properties of object class and complexity is determined by the cardinality of it sets of properties. The number of methods is a measure of class definition as well as being attributes of a class, as attribute are nothing but properties. If all the method complexity are considered as unity then the WMC=n, the no of methods.

k. Depth of inheritance tree (DIT)
It measures the depth of inheritance of the class. If there are multiple inheritances then DIT will be maximum length from node to root of tree [6, 16]. The deeper a class in the hierarchy, greater the number of methods, it is likely to inherit, making it more complex to predict. Deeper the tree greater the design complexity and so does greater potential to reuse of inherited methods. As, it can be very well predicted from the tree diagram in figure 2.

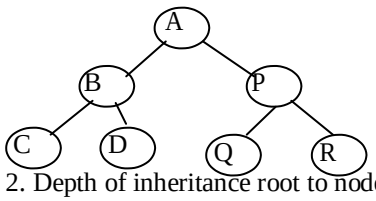

Fig 2. Depth of inheritance root to node

l. Number of Children (NOC)
It is the number of immediate subclasses subordinated to a class in the hierarchy. It measure how many subclasses are going to inherit the methods of parent class [2]. Greater the number of children, greater the reuse and likelihood of improper abstraction of parent class. It will require more testing methods in that class.

m. Coupling between the object classes (CBO)
CBO for a class is the count of number of other classes to which it is coupled. It relates to the notion that as object s cou pled to another if one of them acts on the other. Two classes are coupled when method declared in one class use methods or instance variable defined by other class. The more independent a class is

the easier to reuse it in another application. In order to improve modularity and promote encapsulation, inter object class couples should be kept to a minimum.

n. Response for a class (RFC)

The response set for the class can be expressed as: $RS=\{M\} U_{all\ i} \{R_i\}$

Where R_i is set of methods by method i and {M} is set of all methods in the class.

The response set of a class is a set of methods that can potentially be executed in response to a message received by an object of that class. The cardinality of this ser is a measure of the attributes of objects in the class. It is also a measure of the potential communication between the class and other classes.

o. Lack of cohesion methods (LCOM*)*

LCOM is a count of the number of methods pairs whose similarity is 0 minus the count of methods pairs whose similarity is not zero. The larger the number of similar methods, the more cohesive the class. This uses the notion of degree of similar methods. The degree of similarity for two methods M_1 and M_2 in a class is given by $\sigma() = \{I_1\} \cap \{I_2\}$ where $\{I_1\}$ and $\{I_2\}$ are set of instance variables used by methods M_1 and M_2.

IV. Conclusion

In this paper we have discussed the basic questions about software metrics: why measuring, what to measure, how to measure and when to measure. With the rapid advance of software, their metrics have also developed quickly. Software metrics become the foundation of the software management and essential to the success of software development.We have analyzed the attributes of software, distinguishing attributes of process, product and project. Among all the attributes, complexity is probably the most important one and it comprehends many different aspects of software. The complexity of software will directly affect the eligibility, reliability of the software. We have described various metrics, and most of the metrics defined in the literature, have not been validated using the theory of measurement. Some of them have been validated by showing that they are correlated with other metrics: Function Points have a good correlation with size and effort. With more and more people working hard in this field one can expect to see more thorough and matured software metrics in the near future.

REFERENCES

[1] K.K. Aggarwal and Yogesh Singh, Software Engineering, third edition, 2009 reprinted, new age international publisher.

[2] N.E. Fenton and Shari Lawrencde Pfleeger, Software Metrics: Software metrics A Rigorous and practical approach , second edition. Thomson publication.

[3] S. Morasca, Software Measurement: State of the Art and Related Issues slides from the School of the Italian Group of Informatics Engineering, Rovereto, Italy, September 1995.

[4] Weyuker, Evaluating Software Complexity Measures, IEEE Trans. Software Eng., 14(9), 1988, pp. 1357-1365.

[5] N. Fenton, Software Measurement: a Necessary Scientific Basis, IEEE Trans. Software Eng., 20, 1994, pp. 199-206.

[6] S. Chidamber, C. Kemerer, A Metrics Suite for Object Oriented Design, IEEE Trans. Software Eng., 20/6), 1994, pp. 263-265

[7] Albrecht and J. Gaffney: Software Function, Source Lines of Code, and Development Effort Prediction: A Software Science Validation; in IEEE Trans Software Eng., 9(6), 1983, pp. 639-648

[8] Kumar, A., Kumar, R. and Grover, P. S, "Towards a unified frame work for cohesion measurement in aspect oriented system", Australian conference on software engineering 2008, IEEE, Washington march 2009 pp 57-65.

[9] Thomas J. McCabe, "A Complexity Measure", IEEE transactions on software engineering, IEEE, Washington, Oct 1976, pp 308 320.

[10] N. E. Fenton, "Software Metrics: Successes, Failures & New Directions, " presented at ASM 99: Applications of Software Measurement, S a n J o s e , C A , 1999 .

[11] S. S. Stevens, "On the Theory of Scales of Measurement," Science, vol.103, pp. 677-680. IEEE, "IEEE Std. 1061-1998,

[12] "Standard for a Software Quality Metrics Methodology, revision." Piscataway, NJ,: IEEE Standards Dept., 1998.

[13] L. Briand, S. Morasca, V. Basili, Property-Based Software Engineering Measurement, IEEE Trans. Software Eng. 22(1), 1996, pp. 68-85.

[14] B. Henderson-Sellers, Object Oriented Metrics: Measures of Complexity, Prentice Hall, Upper Saddle River, NJ, 1996

[15] M. Evangelist, Software Complexity Metric Sensitivity to Program Structuring Rules, Journal of Systems and Software, 3(3), 1983, pp. 231-243.

[16] "An Evaluation of the MOOD Set of Object-Oriented Software Metrics", Rachel Harrison, Steve J. Counsell, IEEE transactions on software engineering, vol. 24, no. 6, June 1998.

[17] F. Brito Abreu and W. Melo, "Evaluating the Impact of OO Design on Software Quality," *Proc. Third International Software Metrics Symp.*, Berlin,1996.

[18] "Empirical Analysis of CK Metrics for Object-Oriented Design Complexity: Implications for Software Defects" Ramanath Subramanyam and M.S.Krishnan IEEE transactions on software engineering, vol. 29, no. 4, April 2003

[19] "Applicability of Three Complexity Metrics" D. I. De Silva, N. Kodagoda , H. Perera' The International Conference on Advances in ICT for Emerging Regions - ICTer 2012: 082-088

[20] "The Research on Software Metrics and Software Complexity Metrics" Tu Honglei, Sun Wei, Zhang Yanan, International Forum on Computer Science-Technology and Applications.